Application Report
SPRACN6–July 2019

Fast Integer Division – A Differentiated Offering From C2000™ Product Family

Prasanth Viswanathan Pillai, Himanshu Chaudhary, Aravindhan Karuppiah, Alex Tessarolo

ABSTRACT

This application report provides an overview of the different division and modulo (remainder) functions and its associated properties. Later, the document describes how the different division functions can be implemented using the C28x ISA and intrinsics supported by the compiler.

Contents

1 Introduction 2
2 Different Division Functions 2
3 Intrinsic Support Through TI C2000 Compiler 4
4 Cycle Count 6
5 Summary 6
6 References 6

List of Figures

1 Truncated Division Function 2
2 Floored Division Function 3
3 Euclidean Division Function 3

List of Tables

1 Example Outputs 4
2 Modulo Properties 4
3 Intrinsics Supported and Cycles Using Fast Integer Division Unit on C28x 5
4 Performance Improvement Comparison 6

Trademarks

1 Introduction

Present day processing unit (CPU) used in real time MCUs implement a host of different functions in hardware to reduce the latency and improve the performance. Among these, division and modulo (remainder) are two complex functions to implement. To make matters hard, there are multiple definitions for division and modulo function according to the programming language and computer science literature. These different definitions provide different mathematical properties that can be beneficially employed in the application context. C28x CPU has added new instructions to enable applications to implement different division and modulo functions efficiently. By doing so, C28x CPU became the first CPU used in the MCU application space to implement these operations in hardware.

Low latency, ability to interrupt and higher efficiency are some of the important considerations when designing the instruction set architecture for the CPU of a real time MCU. The inputs from the real world which are used for processing can be of different types (unsigned, signed) and different sizes (16, 32, 64, 128, and so forth). The instruction set architecture should enable seamless processing of different combination of these values also. The new instructions used to enable integer division are interruptible, have very low latency and support different types of operations (ui32/ui32, i32/ui32, i64/i32, ui64/ui32, ui64/ui64, i64/i64, and so forth).

2 Different Division Functions

Programming language and computer science literature mainly defines three different division and modulo functions. This section describes the different division and modulo function languages, and their salient properties.

2.1 Truncated Division or Traditional Division

Truncated division is natively used in many programming languages including C and it is the most commonly used division function. The truncated division function is defined as follows:

Quotient = *trunc*(Numerator/Denominator)

Remainder = Numerator – Quotient*Denominator

The transfer function of the truncated division is shown in Figure 1. In this definition, the remainder will always have the sign of the numerator. Here, the function is non periodic since there is a “platform” around zero point. Due to the non-linearity around zero point, the function is not preferred in control algorithms.

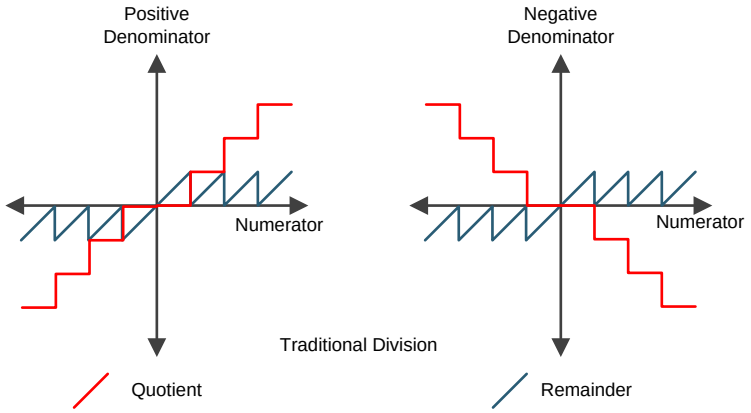

Figure 1. Truncated Division Function

2.2 Floored Division or Modulo Division

Floored division is defined as follows

Quotient = *floor*(Numerator/Denominator)
Remainder = Numerator – Quotient*Denominator

The function is defined such that the remainder will always have the sign of denominator. The function is linear around the zero point. The transfer function of the floored division is provided in Figure 2. This division function is more regular compared to the traditional division function.

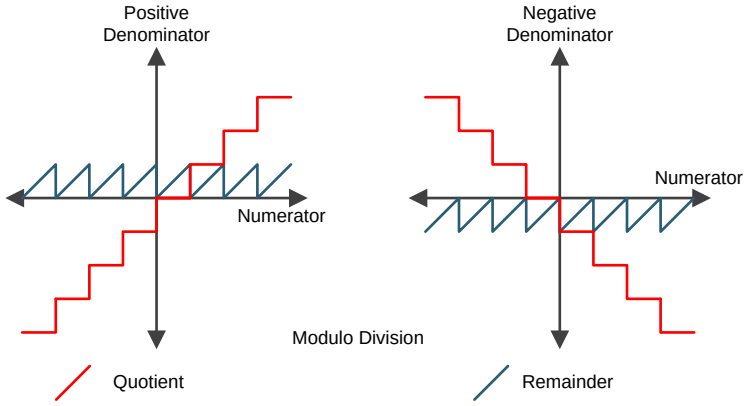

Figure 2. Floored Division Function

2.3 Euclidean Division

This form of division is derived from Euclid's theorem. The quotient and remainder results are defined as follows:

if (Denominator > 0)
  Quotient = *floor*(Numerator/Denominator)
else if(Denominator < 0)
  Quotient = *ceil*(Numerator/Denominator)

The transfer function of Euclidean division is shown in Figure 3. The remainder is always positive in Euclidean division. The division function is linear around the zero point and the modulo function is periodic. Due to its unique properties, it is preferred for implementing several algorithms.

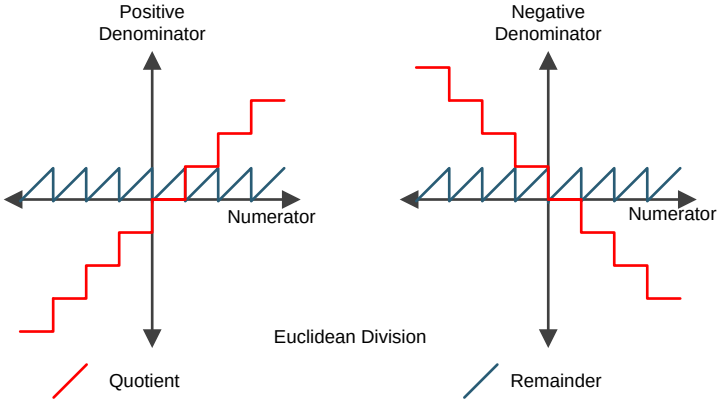

Figure 3. Euclidean Division Function

The quotient and remainder results obtained for different types of division for a set of sample input values are given in Table 1.

Table 1. Example Outputs

Numerator	Denominator	Traditional Division		Floored Division		Euclidean Division	
		Quotient	modulo	Quotient	modulo	Quotient	modulo
7	4	1	3	1	3	1	3
-7	4	-1	-3	-2	1	-2	1
7	-4	-1	3	-2	-1	-1	3
-7	-4	1	-3	1	-3	2	1

The properties of the remainder (modulo) operation for different division types are provided in Table 2 [1].

Table 2. Modulo Properties

Sl No	Property	Traditional Division	Floored Division	Euclidean Division
1	Periodicity		X	X
2	Regularity	Low	Medium	High
3	Preservation of Numerator Sign	X		
4	Preservation of Denominator Sign		X	
5	Non-negative unique representation			X

3 Intrinsic Support Through TI C2000 Compiler

The division functions are standardized operations and hence to provide optimal cycles and ease of developing applications using these operations, TI C2000 compiler provides support through intrinsics. The *TMS320C28x Optimizing C/C++ Compiler v18.12.0.LTS User's Guide* [2] and later versions support the generation of division functions in one of three ways:

- Intrinsics, declared in stdlib.h, which take numerator and denominator and return a structure containing both the remainder and quotient. The intrinsics supported are mentioned in the *TMS320C28x Optimizing C/C++ Compiler v18.12.0.LTS User's Guide*.
- Operators for division (/) and modulo (%), which will automatically be optimized.
- Standard library functions ldiv and lldiv, found in stdlib.h.

Euclidean/Modulo division are supported only using intrinsics. Operators and the standard library functions will perform traditional division according to the C standard.

Compiler option, --idiv_support, controls support for these division types. A value of 'idiv0' implies hardware acceleration for different division functions and value of none implies no hardware acceleration for the different division functions. The option is only valid when FPU32 or FPU64 is available (--float_support=fpu32 or fpu64) and when using the C2000 EABI (--abi=eabi).

For more details on intrinsics definitions, macros, and additional background information, see the *TMS320C28x Optimizing C/C++ Compiler v18.12.0.LTS User's Guide* and the *TMS320C28x Assembly Language Tools v18.12.0.LTS User's Guide User's Guide* [3].

3.1 Software Examples

To ease development of applications using the Fast Integer Division (FID) unit, TI also provides demo examples inside the library section of C2000WARE (libraries\math\FASTINTDIV) [4] to showcase the usage of various integer division intrinsics. This example provides users with the usage of each of the 21 intrinsics. The cycles can be measured to see the significant gain provided by using the TI's FID unit. The entire process of using the acceleration provided by the FID unit has been made very simple through the intrinsics and example provided. Table 3 provides the list of intrinsics and cycles.

Table 3. Intrinsics Supported and Cycles Using Fast Integer Division Unit on C28x

Division Type	Intrinsic	Cycles
16-bit by 16-bit traditional	__traditional_div_i16byi16() or int/int	16
16-bit by 16-bit Euclidean	__euclidean_div_i16byi16()	14
16-bit by 16-bit Modulo	__modulo_div_i16byi16()	14
16-bit by 16-bit traditional unsigned	__traditional_div_u16byu16() or uint16_t/uint16_t	14
32-bit by 32-bit traditional	__traditional_div_i32byi32() or long/long	13
32-bit by 32-bit Euclidean	__euclidean_div_i32byi32()	14
32-bit by 32-bit Modulo	__modulo_div_i32byi32()	14
32-bit by 32-bit traditional - long/unsigned long	__traditional_div_i32byu32() or long/unsigned long	14
32-bit by 32-bit Modulo	__modul0_div_i32byu32()	14
32-bit by 32-bit traditional - unsigned long/unsigned long	__traditional_div_u32byu32() or unsigned long/unsigned long	12
32-bit by 16-bit traditional	__traditional_div_i32by16() or long/int	18
32-bit by 16-bit Euclidean	__euclidean_div_i32bi16()	16
32-bit by 16-bit Modulo	__modulo_div_i32byi16()	16
32-bit by 16-bit traditional unsigned long/unsigned int	__traditional_div_u32byu16() or unsigned long/uint16_t	13
64-bit by 64-bit traditional	__traditional_div_i64byi64() or long long/long long	42
64-bit by 64-bit Euclidean	__euclidean_div_i64byi64()	42
32-bit by 64-bit Modulo	__modulo_div_i64byi64()	42
64-bit by 64-bit traditional - long long/unsigned long long	__traditional_div_i64byu64() or long long/unsigned long long	42
64-bit by 64-bit Modulo	__euclidean_div_i64byu64()	42
64-bit by 64-bit Modulo	__modulo_div_i64byu64()	42
64-bit by 64-bit traditional - unsigned long long/unsigned long long	__traditional_div_u64byu64() or unsigned long long/unsigned long long	42

4 Cycle Count

The cycles for the different types of division operations and sizes of the operands are listed below. These can be profiled using the examples provided in the C2000WARE as well. Wide variety of division operations, varying operands sizes are listed below along with cycles numbers. The boost in cycles using the fast integer division is shown in Table 4 with respect to the cycles needed to do the same operation on the C28x CPU.

Table 4. Performance Improvement Comparison

Division Operation	Using C Operator '/' Without FASTINTDIV Hardware on C28x	Using Intrinsics With FASTINTDIV Hardware + C28x	Improvement Factor
i16/i16 traditional	52	16	3.3
i16/i16 Euclidean	56	14	4.0
i16/i16 Modulo	56	14	4.0
u16/u16	56	14	4.0
i32/i32 traditional	59	13	4.5
i32/i32 Euclidean	63	14	4.5
i32/i32 Modulo	63	14	4.5
i32/u32 traditional	37	14	2.6
i32/u32 Modulo	41	14	2.9
u32/u32	37	12	3.1
i32/i16 traditional	60	18	3.3
i32/i16 Euclidean	64	16	4.0
i32/i16 Modulo	64	16	4.0
u32/u16	38	13	2.9
i64/i64 traditional (1)	78-2631	42	1.9-62.6
i64/i64 Euclidean (1)	82-2635	42	2.0-62.7
i64/i64 Modulo (1)	82-2635	42	2.0-62.7
i64/u64 traditional (1)	54-2605	42	1.3-62.0
i64/u64 Euclidean (1)	58-2609	42	1.4-62.1
i64/u6 Modulo (1)	58-2609	42	1.4-62.1
u64/u64/ (1)	53-2548	42	1.3-60.7

(1) The FASTINTDIV hardware implements 64-bit integer division with optimal fixed number of cycles for fast deterministic behavior. MCUs without such hardware acceleration, implement 64-bit integer division using generic CPU instructions that are not optimized for division or use algorithm techniques that optimize execution based on the value of the numerator and denominator. For instance, if the value of the numerator and denominator is less than 32-bits the software will execute a 32-bit division. Hence, the number of cycles can vary significantly and for large numerator and denominator values, overall cycles are much higher than achievable by the FASTINTDIV accelerator

5 Summary

The differentiation provided by the Texas Instruments Fast Integer Division (FID) unit reduces the latency of different division operations. The FID unit provides several folds improvement in performance for different types of division operations. The cycles saved for division operation along with the many of the other differentiations provided in C2000 devices such as FPU, TMU and the inherent C28x DSP minimizes the latency of control loop calculations and opens up usage in new applications. The intrinsics support in compiler ensures that the use of this differentiation is just a few additional lines of code.

6 References

1. R. T. Boute, “The euclidean definition of the functions div and mod,” ACM Transactions on Programming Languages and Systems (TOPLAS), 1992.
2. Texas Instruments: *TMS320C28x Optimizing C/C++ Compiler v18.12.0.LTS User's Guide*
3. Texas Instruments: *TMS320C28x Assembly Language Tools v18.12.0.LTS User's Guide User's Guide*
4. C2000Ware for C2000 MCUs (http://www.ti.com/tool/C2000WARE)

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265